Noname manuscript No.
(will be inserted by the editor)

Compressed String Dictionary Search with Edit Distance One

Djamal Belazzougui · **Rossano Venturini**

the date of receipt and acceptance should be inserted later

Abstract In this paper we present different solutions for the problem of indexing a dictionary of strings in compressed space. Given a pattern P, the index has to report all the strings in the dictionary having *edit distance* at most one with P. Our first solution is able to solve queries in (almost optimal) $O(|P| + occ)$ time where occ is the number of strings in the dictionary having edit distance at most one with P. The space complexity of this solution is bounded in terms of the k-th order entropy of the indexed dictionary. A second solution further improves this space complexity at the cost of increasing the query time. Finally, we propose randomized solutions (Monte Carlo and Las Vegas) which achieve simultaneously the time complexity of the first solution and the space complexity of the second one.

1 Introduction

Modern web search, information retrieval, data base and data mining applications often require solving string processing and searching tasks. Most of such tasks boil down to some basic algorithmic primitives which involve a large dictionary of variable-length strings. Solving approximate searches over dictionaries of strings is an important primitive that appears frequently in many practical scenarios. In Web search, for

The work is an extended version of the paper [8] appeared in Proceedings of 23rd Annual Symposium on Combinatorial Pattern Matching, 2012. This work has been partially supported by Academy of Finland under grant 250345 (CoECGR), the French ANR-2010-COSI-004 MAPPI project, PRIN ARS Technomedia 2012, the Midas EU Project, Grant Agreement no. 318786, and the eCloud EU Project, Grant Agreement no. 325091.

Djamal Belazzougui
Helsinki Institute for Information Technology HIIT, Department of Computer Science, University of Helsinki, Finland.

Rossano Venturini
Department of Computer Science, University of Pisa, Italy.

E-mail: djamal.belazzougui@cs.helsinki.fi, rossano.venturini@unipi.it

example, users query the engine with possibly misspelled terms that can be corrected by choosing among the closest terms stored in a trustable dictionary. In data mining and data base applications, instead, an automatically built dictionary may contain noise in the form of misspelled strings. Thus, we may need to resort to approximate searches in order to identify the closest dictionary strings with respect to a (correct) input string.

The *edit distance* (also known as Levenstein distance) is the most commonly used distance to deal with misspelled strings. The edit distance between two strings is defined as the minimal number of edit operations required to transform the first string into the second string. There are three possible edit operations: deletion of a symbol, insertion of a symbol and substitution of a symbol with another.

The problem *string dictionary search with edit distance one* is defined as follows. Let $D = \{S_1, S_2, \ldots, S_d\}$ be a dictionary of d strings of total length n drawn from an alphabet $\Sigma = [\sigma]$. We want to build a (compressed) index that, given any string $P[1,p]$, reports all the strings in D having edit distance at most 1 with P. For simplicity, we assume that the strings in D are all distinct and sorted lexicographically (namely, for any $1 \leq i < d$, $S_i < S_{i+1}$).

In this paper we provide two compressed solutions for the problem. The first solution guarantees (almost) optimal query time while requiring compressed space. Namely, we show how to obtain an index of $2nH_k + n \cdot o(\log \sigma) + 2d \log d$ bits, that is able to report all the occ strings having edit distance at most 1 with P in $O(p + occ)$ time. Here H_k denotes the k-th order entropy of the strings in the dictionary for any fixed $k = o(\log_\sigma n)$. Interestingly, the time complexity of this solution is independent of alphabet size. This is quite an uncommon result for compressed data structures dealing with texts. The second solution provides space/time tradeoffs by using a completely different approach. Its space occupancy, indeed, decreases to $nH_k + n \cdot o(\log \sigma)$ bits. This better space bound is obtained at the cost of increasing the query time to $O(p \log\log \sigma + occ)$.

Interestingly, our first solution can be easily extended to support an additional operation which has interesting practical applications. Assume that each string S_i in D has been assigned a score $c(S_i)$, which, for example, could establish the relative importance of any string with respect to the others. The first solution can be extended to support the extra operation $\mathsf{Top}(P[1,p],k)$ that reports the k highest scored strings in D having edit distance at most one with P. This operation is solved in $O(p + k \log k)$ time and returns the occurrences sorted by their scores.

We finally show how to introduce randomization in these solutions to derive Monte Carlo and Las Vegas solutions. These solutions are able to either reduce the space occupancy or improve the query time of the deterministic solutions.

2 Related work

The literature presents several solutions to the problem of indexing string dictionaries to efficiently search with error distance one.

The theoretical study of the problem has been initiated by Yao and Yao in [34]. They present a solution in the bit probe model for the related problem of indexing a

dictionary of strings supporting searches with Hamming distance one. Their solution indexes d binary strings of length m each (i.e., $n = dm$) by using $O(n \log m)$ bits and solves a query with $O(m \log\log d)$ bit probes. Subsequently, Brodal and Venkatesh [12] improve the above time complexity by a factor $\log\log d$: their solution uses $O(n \log d)$ bits of space and $O(m)$ bit probes. In the RAM model, the time complexity is $O(m/w)$, where w is the size of a machine word.

Brodal and Gąsieniec [11] propose two solutions to solve searches with Hamming distance one[1]. The first solution reports all the *occ* strings having Hamming distance at most one with P in time $O(p + occ)$ by using $O(\sigma \cdot n \log n)$ bits of space. The main data structure is a trie that indexes strings in D plus extra strings. An extra string is a string that does not belong to D but has Hamming distance one with at least a string in D. Clearly, each root-to-leaf path in the trie represents either a string in D or an extra string. The leaf representing a string S is associated with the list of indices of strings in D having Hamming distance one with S. The query for P is solved by navigating the trie. If a leaf is reached, the list of indices stored in the leaf is reported. Construction time and space occupancy for non-constant size alphabets are the major drawbacks of this solution. Indeed, it is unknown how to build this data structure in $O(n\sigma)$ time or use $o(n\sigma \log n)$ bits of space. Their second solution is slower than the previous one by an additive term $O(\log d)$ (namely, query time is $O(p + \log d + occ)$) [11]. The advantage is represented by its space occupancy which is $O(n \log n)$ and, thus, it is better for non-constant size alphabets[2]. The solution resorts to two tries augmented with a sorted list of identifiers. These tries index, respectively, the strings in D and the strings in D reversed. The query algorithm exploits the following property: if there exists a string S in D having distance one with $P[1,p]$, it can be factorized as $S = P[1,i] \cdot c \cdot P[i+2,p]$, for some index i and symbol $c \in \Sigma$. This is a key property that has been exploited by almost all the subsequent solutions, including ours. These solutions differ from each other in the data structures and the algorithms they use to discover all these factorizations. For each string $S[1,s]$ in D, Brodal and Gąsieniec consider all the triplets $(\mathsf{np}_i(S), S[i+1], \mathsf{ns}_{i+2}(S))$, where $\mathsf{np}_i(S)$ is the identifier of the node corresponding to prefix $S[1,i]$ in the first trie and $\mathsf{ns}_{i+2}(S)$ is the identifier of the node corresponding to $S[i+2,s]$ reversed in the second trie. It is easy to index these triples by inserting them in a search tree that is able to report, given a pair of node identifiers u and v, all the triples with u in the first component and v in the third component. This is the core operation of an algorithm solving any query in $O(p \log n + occ)$ time. For any index i, the algorithm identifies the nodes $\mathsf{np}_i(P)$ and $\mathsf{ns}_{i+2}(P)$ and uses the search tree by querying for these two nodes. If the triple $(\mathsf{np}_i(P), c, \mathsf{ns}_{i+2}(P))$ is returned, then the string $S = P[1,i]c \cdot P[i+2,p]$ is in D and has distance one from P. The above query time can be further improved by replacing the search tree with a sorted list of string identifiers in each node of the reverse trie and by resorting to (a sort of) fractional cascading during the query resolution.

The current best solution for the string dictionary search with edit distance one problem has been presented by Belazzougui [4]. This solution follows a similar ap-

[1] However, they can be easily extended to deal with the more general edit distance.

[2] Actually, the paper [11] described only a solution for binary alphabet. However, it is not hard to obtain the claimed space and time complexities also for non-constant alphabet sizes.

proach but obtains significantly better time and space complexities. Indeed, this solution achieves $O(p+occ)$ query time by requiring optimal $O(n\log\sigma)$ bits of space. This is obtained by carefully combining compact tries, (minimal) perfect hash functions and Karp-Rabin signatures.

Finally, we observe that this problem can be seen as a simpler instance of either approximate full-text indexing or approximate dictionary matching with one or more errors. However, currently best solutions for these more general problems are not competitive with the solutions presented in this paper (see e.g., [1, 5, 15]).

3 Background

In this section we collect a set of algorithmic tools that will be used in our solutions. In the following we report each result together with a brief description of the solved problem. More details can be obtained by consulting the corresponding references. All the results hold in the unit cost word-RAM model, where each memory word has size w bits.

Empirical Entropy. Let $T[1,n]$ be a string drawn from the alphabet $\Sigma = \{a_1,\ldots,a_h\}$. For each $a_i \in \Sigma$, we let n_i denote the number of occurrences of a_i in T. The zero-th order *empirical* entropy of T is defined as follows.

$$H_0(T) = \frac{1}{|T|}\sum_{i=1}^{h} n_i \log\frac{n}{n_i} \tag{1}$$

Note that $|T|H_0(T)$ provides an information-theoretic lower bound for the output size of any compressor that encodes each symbol of T with a fixed code [33].

For any string w of length k, we denote by w_T the string of single symbols following the occurrences of w in T, taken from left to right. For example, if $T =$ `mississippi` and $w =$ `si`, we have $w_T =$ `sp` since the two occurrences of `si` in T are followed by the symbols `s` and `p`, respectively. The k-th order *empirical* entropy of T is defined as follows.

$$H_k(T) = \frac{1}{|T|}\sum_{w\in\Sigma^k} |w_T|\, H_0(w_T) \tag{2}$$

We have $H_k(T) \geq H_{k+1}(T)$ for any $k \geq 0$. The term $|T|H_k(T)$ is an information-theoretic lower bound for the output size of any compressor that encodes each symbol of T with a code that depends on the symbol itself and on the k immediately preceding symbols [26].

Compressed strings with fast random access. In the following we will require the availability of a storage scheme for strings that uses compressed space still being able to access in $O(1)$ time any symbol of the represented string T. To this aim, we use the following result [21].

Lemma 1 *Given a text $T[1,n]$ drawn from an alphabet $\Sigma = [\sigma]$, $\sigma \leq n$, there exists a compressed data structure that supports the access in constant time of any substring of T of length $O(\log n)$ bits requiring $nH_k(T) + \rho$ bits, where $H_k(T)$ denotes the k-th empirical entropy of T and $k = o(\log_\sigma n)$. The redundancy ρ depends on the alphabet size σ: $\rho = o(n)$ if $\log\sigma = o(\sqrt{\log n/k})$, $\rho = n \cdot o(\log\sigma)$ otherwise.*

The scheme can be also used in cases in which T is the concatenation of a set of strings (namely, $T = S_1 \cdot S_2 \cdot \ldots \cdot S_d$). The starting positions of strings in T are stored by resorting to Elias-Fano's representation [17, 18] within $d\log(\frac{n}{d}) + O(d)$ bits. This additional structure allows us to access an arbitrary portion of any string in optimal time.

Karp-Rabin signature. Given a string $S[1,s]$, the Karp-Rabin signature function [25] $\mathsf{kr}(S)$ is equal to $\sum_{i=1}^{s} S[i] \cdot t^i \pmod M$, where M is a prime number and t is a randomly chosen integer in $[1, M-1]$. Given a dictionary of strings D containing d strings of total length n, we can obtain an instance kr of the Karp-Rabin function that maps strings in D to the integers in $[M-1]$ without collisions, with M chosen among the first $O(n \cdot d^2)$ integers. It is known that a value of t that guarantees injectivity can be found with constant number of attempts in expectation (see the analysis in [16]). Notice that the representation of kr requires $O(\log n + \log d) = O(\log n)$ bits of space.

Interestingly, the Karp-Rabin function guarantees that, after a preprocessing phase over a string S, signatures of strings close enough to S can be computed in constant time. This property is formally stated by the following lemma.

Lemma 2 *Given a string $S[1,s]$, for every prefix P of S, $\mathsf{kr}(P)$ can be computed in constant time. Moreover, for every string Q at distance 1 from S, $\mathsf{kr}(Q)$ can be computed in constant time, after a preprocessing phase that takes $O(s)$ time.*

Minimal perfect hash function. Given a subset of $S = \{x_1, x_2, \ldots, x_n\} \subseteq U = [2^w]$ of size n, a minimal perfect hash function has to injectively map keys in S to the integers in $[n]$. Hagerup and Tholey [24] show how to build a space/time optimal minimal perfect hash function as stated by the following lemma.

Lemma 3 *Given a subset of $S \subseteq U = [2^w]$ of size n, one can construct in $O(n)$ time a minimal perfect hash function for S that can be evaluated in $O(1)$ time and requires $n\log e + o(n)$ bits of space.*

Compressed static function. Often keys in $S = \{x_1, x_2, \ldots, x_n\}$ have associated satellite data (called *values*) from an alphabet $\Sigma = [\sigma]$ and we are asked to build a dictionary that, given a key $x \in S$, returns its associated value. An arbitrary value is returned whenever $x \notin S$. Essentially, we are defining a static function F whose domain is the set S and whose codomain is formed by the values associated with those keys (i.e., $\Sigma = \{F(x_1), \ldots, F(x_n)\}$). The problem asks to evaluate F on its domain with the possibility of returning any value for keys in $U \setminus S$. Often the values associated with the keys follow a skewed distribution: few values are considerably more frequent than others. In these scenarios, it is desirable to achieve space that depends on the entropy of the data rather than on the number of possible values. Thus, designing *compressed*

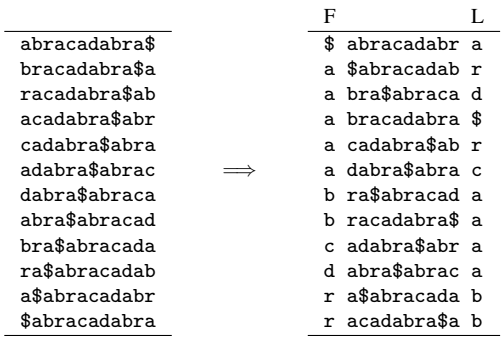

Fig. 1 Example of Burrows-Wheeler transform for the string $T = \texttt{abracadabra\$}$. The matrix on the right has the rows sorted in lexicographic order. The output of the Bwt is the column $L = \texttt{ard\$rcaaaabb}$.

static functions asks to represent F with constant evaluation time by using space close to nH_0 bits, where H_0 denotes the 0-th order empirical entropy of the sequence $F(x_1), F(x_2), \ldots, F(x_n)$. The current best result for this problem [9] is reported in the following theorem.

Theorem 1 *A static function F defined over a subset of keys $S = \{x_1, \ldots, x_n\} \subseteq U = [2^w]$ into values drawn from an alphabet $\Sigma = [\sigma]$, $\sigma \leq n$ can be represented with $nH_0 + O(\frac{n(\log\log n + H_0)\log\log n}{\log n}) + O(\sigma + \log w)$ bits of space so that evaluating $F(x)$ for any x takes constant time, where H_0 denotes the empirical zero-th order entropy of the sequence $F(x_1), \ldots, F(x_n)$. The scheme can be built in $O(n)$ expected time.*

Approximate membership. An approximate membership data structure (*AM* for short) stores an approximate representation of a set $S = \{x_1, x_2, \ldots, x_n\} \subseteq U = [2^w]$. The representation is approximate in the following sense: the query on an element $x \in U$ always returns true if $x \in S$, false with probability at least $1 - \varepsilon$ if $x \in U \setminus S$, where the real parameter $\varepsilon \in (0,1)$ (called *false positive probability*) is specified at construction time. The Bloom filter [10] is the most popular approximate membership data structure, but only more recent data structures [9, 14, 30] are known to have optimal time and space complexities (up to constant factors).

Lemma 4 *Given a set $S = \{x_1, \ldots, x_n\} \subseteq U = [2^w]$ and a parameter ε such that $0 < \varepsilon < 1$, there exists an approximate membership data structure for the set S with false positive probability ε requiring $O(n\log(1/\varepsilon))$ bits of space and answering any query in constant time.*

Burrows-Wheeler transform. In 1994 Burrows and Wheeler [13] introduced a new compression algorithm based on a reversible transformation, now called the *Burrows-Wheeler Transform* (Bwt from now on). The Bwt transforms the input string T into a new string that is easier to compress. The Bwt of T, hereafter denoted by Bwt_T, is built with three basic steps (see Figure 1):

1. append to T a special symbol \$ smaller than any other symbol of Σ;

2. form a *conceptual* matrix M_T whose rows are the cyclic rotations of string $T\$$ in lexicographic order;
3. construct string L by taking the last column of the sorted matrix M_T. We set $\mathsf{Bwt}_T = L$.

Every column of M_T, hence also the transformed string L, is a permutation of $T\$$. In particular the first column of M_T, call it F, is obtained by lexicographically sorting the symbols of $T\$$ (or, equally, the symbols of L). Note that sorting the rows of M_T is essentially equivalent to sorting the suffixes of T, because of the presence of the special symbol \$. This shows that: (1) symbols following the same substring (*context*) in T are grouped together in L, and thus give raise to clusters of nearly identical symbols; (2) there is an obvious relation between M_T and the suffix array SA_T of T. Property 1 is the key for devising modern data compressors (see e.g. [26]), Property 2 is crucial for designing compressed indexes (see e.g. [19, 28]) and, additionally, suggests a way to compute the Bwt through the construction of the suffix array of T: $L[1] = T[n]$ and, for any $2 \leq i \leq n$, set $L[i] = T[SA_T[i]-1]$.

Burrows and Wheeler [13] devised two properties for the invertibility of the Bwt:

(a) Since the rows in M_T are cyclically rotated, $L[i]$ *precedes* $F[i]$ in the original string T;
(b) For any $c \in \Sigma$, the ℓ-th occurrence of c in F and the ℓ-th occurrence of c in L correspond to the *same* character of the string T.

As a result, the original text T can be obtained backwards from L by resorting to function LF (also called Last-to-First column mapping or LF-mapping) that maps row indexes to row indexes, and is defined as:

$$LF(i) = C[L[i]] + \mathsf{rank}_{L[i]}(L,i),$$

where $C[L[i]]$ counts the number of occurrences in T of symbols smaller than $L[i]$ and $\mathsf{rank}_{L[i]}(L,i)$ is a function that returns the number of times symbol $L[i]$ occurs in the prefix $L[1,i]$. We talk about LF-mapping because the symbol $c = L[i]$ is located in the first column of M_T at position $LF(i)$. The LF-mapping allows one to navigate T backwards: if $T[k] = L[i]$, then $T[k-1] = L[LF(i)]$ because row $LF(i)$ of M_T starts with $T[k]$ and thus ends with $T[k-1]$. In this way, we can reconstruct T backwards by starting at the first row, equal to $\$T$, and repeatedly applying LF for n steps. As an example, see Figure 1 in which the 3rd `a` in L lies onto the row which starts with `bracadabra$` and, correctly, the 3rd `a` in F lies onto the row which starts with `abracadabra$`. That symbol `a` is $T[1]$.

Compressed full-text indexing. Ferragina and Manzini [20] show that data structures supporting rank queries on the string L suffice to search for an arbitrary pattern $P[1,p]$ as a substring of the indexed text T. For any $i \in [1,|L|]$ and $c \in \Sigma$, the query $\mathsf{rank}(i,c)$ on L returns the number of occurrences of symbol c in the prefix $L[1,i]$. The resulting search procedure is now called *backward search* and is illustrated in Figure 2. It works in p phases, each preserving the invariant: *At the end of the i-th phase,* $[\mathsf{First},\mathsf{Last}]$ *is the range of contiguous rows in* M_T *which are prefixed by* $P[i,p]$. Backward_search

Algorithm Backward_search($P[1,p]$)

1. $i = p$, $c = P[p]$, First $= C[c]+1$, Last $= C[c+1]$;
2. **while** ((First $\leq$ Last) **and** ($i \geq 2$)) **do**
3. $\quad c = P[i-1]$;
4. $\quad$ First $= C[c] + \mathsf{rank}_c(L,$ First $-1)+1$;
5. $\quad$ Last $= C[c] + \mathsf{rank}_c(L,$ Last$)$;
6. $\quad i = i-1$;
7. **if** (Last $<$ First) **then return** "no rows prefixed by P" **else return** [First, Last].

Fig. 2 The algorithm to find the range [First, Last] of rows of M_T prefixed by $P[1,p]$.

starts with $i = p$ so that First and Last are determined via the array C (step 1). Ferragina and Manzini proved that the pseudo-code in Figure 2 maintains the invariant above for all phases, so [First, Last] delimits at the end the rows prefixed by P (if any). Steps 4 and 5 are the dominant costs of each iteration of Backward_search. They are computed efficiently by using appropriate data structures. Array C is small and occupies $O(\sigma \log n)$ bits. Efficiently supporting rank queries over Bwt requires more sophisticated data structures. The literature offers many theoretical and practical solutions for this problem (see e.g., [2, 3, 6, 19, 28] and references therein). The following lemma summarizes the results we use in our solution.

Lemma 5 *Let $T[1,n]$ be a string over alphabet $\Sigma = [\sigma]$, $\sigma \leq n$, $L = \mathsf{Bwt}_T$ be its Burrows-Wheeler transform and w be the size of a memory word.*

1. *For $\sigma = O(\mathtt{poly}(w))$, there exists a data structure which supports* rank *queries and the retrieval of any symbol of L in constant time, by using $nH_k(T) + o(n)$ bits of space, for any $k \leq \alpha \log_\sigma n$ and $0 < \alpha < 1$.*
2. *For larger σ, there exists a data structure which supports* rank *queries and the retrieval of any symbol of L in $O(\log\log_w \sigma)$ time, by using $nH_k(T) + o(n)(1 + H_k(T))$ bits of space, for any $k \leq \alpha \log_\sigma n$ and $0 < \alpha < 1$*

By plugging Lemma 5 into Backward_search, we obtain the following theorem.

Theorem 2 *Given a text $T[1,n]$ drawn from an alphabet $\Sigma = [\sigma]$, $\sigma \leq n$, there exists a compressed index that takes $p \times t_{\mathsf{rank}}$ time to support* Backward_search$(P[1,p])$*, where t_{rank} is the time cost of a single* rank *operation over $L = \mathsf{Bwt}_T$. The space occupancy is bounded by $nH_k(T) + \rho$ bits, for any $k \leq \alpha \log_\sigma n$ and $0 < \alpha < 1$.*

The redundancy ρ is $o(n)$ bits and t_{rank} is $O(1)$ when $\sigma = O(\mathtt{poly}(w))$ while $\rho = o(n)(1 + H_k(T))$ bits and $t_{\mathsf{rank}} = O(\log\log_w \sigma)$ otherwise, where w is the size of a memory word.

Notice that compressed indexes support also other operations, like locate and extract, which are slower than Backward_search in that they require $\mathtt{polylog}(n)$ time per occurrence [19, 28]. We do not go into further details on these operations because they are not required in our solution.

4 A hashing-based solution

Our first solution can be seen as a compressed variant of the solution presented in [4]. However, we need to apply significant and non-trivial changes to that solution in order to achieve compressed space and to retain exactly the same (almost optimal) query time. More formally, in this section we prove the following theorem.

Theorem 3 *Given a set $D = \{S_1, S_2, \ldots, S_d\}$ of d strings of total length n drawn from an alphabet $\Sigma = [\sigma]$, $\sigma \leq n$, there exists an index that, given any pattern $P[1,p]$, reports in $O(p + occ)$ worst-case time all the occ strings in D having edit distance at most one with P. The index requires*

1. *$nH_k + o(n) + 2d \log d$ bits of space, if $\sigma = O(1)$;*
2. *$2nH_k + o(n) + 2d \log d$ bits of space, if $\log \sigma = o(\sqrt{\log n / k})$;*
3. *$2nH_k + n \cdot o(\log \sigma) + 2d \log d$ bits of space, otherwise,*

for any fixed $k = o(\log_\sigma n)$.

At a high level our solution works as follows. First, it identifies a set of $O(p + occ)$ *candidate strings* being a superset of the strings that have edit distance at most one with P. Then, it discards all candidate strings that actually do not belong to D. For the moment, let us assume that establishing whether or not a candidate string belongs to D costs constant time. Later, we will discuss how to efficiently perform this non-trivial task[3].

Our solution asks to identify the strings in D that share prefixes and suffixes with the query string P. For this aim we resort to two patricia tries PT and $\mathsf{PT_r}$ that index the strings in D and the strings in D written in reversed order, respectively. Each node in each patricia trie is uniquely identified by the time of its visit in the preorder visit of the tree. The tree structure of each patricia trie is represented in $O(d)$ bits with standard succinct solutions [27]. In order to perform searches on patricia tries, we add data structures to compute the length of longest common prefix (lcp) and longest common suffix (lcp_r) for any pair of strings in D. A standard constant time solution requiring $O(d(1 + \log \frac{n}{d}))$ bits of space is obtained by writing lcps between lexicographically consecutive strings (resp. reverse strings) using Elias-Fano's representation [17, 18] and by resorting to Range Minimum Queries (rmq) (see e.g., [23]) on these arrays. Fast percolation of the tries is obtained by augmenting the branching nodes with monotone minimal perfect hash functions as described in [7]. In this way choosing the correct edge to follow from the current node can be done in constant time regardless of the alphabet size. The extra space cost is bounded by $O(d \log \log \sigma)$ bits. Thus, the representation of the two patricia tries uses $O(d(\log \log \sigma + \log \frac{n}{d}))$ bits. The following fact states that this space occupancy is $O(\frac{n \log \sigma \log \log n}{\log n})$ bits and, thus, within the lower-order terms of Theorem 3.

Fact 1 $d(\log \log \sigma + \log \frac{n}{d}) = O(\frac{n \log \sigma \log \log n}{\log n})$

[3] Notice that just accessing each symbol of these candidate strings would cost $O(p + p \cdot occ)$ time in total which is much higher than our claimed complexity.

Proof Observe that, since all the d dictionary strings are distinct, their average length must be at least $\log d$ bits. This implies that their total length in bits, i.e., $n\log\sigma$ bits, must be at least $d\log d$, i.e., $d\log d \leq n\log\sigma$. Consider now two cases depending on the value of d. In the first case we assume that $d \leq \frac{n}{\log^2 n}$, which gives $d(\log\log\sigma + \log\frac{n}{d}) \leq \frac{n}{\log^2 n}(\log\log n + \log n) = O(\frac{n}{\log n})$, and the result follows. Conversely, in the second case we assume that $d > \frac{n}{\log^2 n}$, which gives $\log(n/d) \leq 2\log\log n$ and $\log d \geq \log n - 2\log\log n$ and, thus, $d(\log n - 2\log\log n) \leq d\log d \leq n\log\sigma$. This allows us to deduce that $d \leq \frac{n\log\sigma}{\log n - 2\log\log n}$ and, thus, $d(\log\log\sigma + \log\frac{n}{d})$ is at most $O(d\log\log n) \leq O(\frac{n\log\sigma}{\log n}\log\log n) = O(\frac{n\log\sigma\log\log n}{\log n})$ as claimed. In what preceded we assumed that $\log\log\sigma \leq \log\log n$. If that was not the case, then $\sigma > n$ and $d\log\log\sigma = O(\frac{n\log\sigma\log\log n}{\log n})$ trivially holds. □

The correctness of the steps performed during the search on the patricia tries is established by comparing the searched string and labels on the traversed edges. This is done by directly accessing the appropriate portion of the strings in D from their compressed representations. For this aim D is represented by resorting to the compressed scheme of Lemma 1 that allows constant time access to any symbol of any string in D. The space required by this is bounded by the k-th order entropy according to Lemma 1. Since the strings do not keep their original order in the trie PT_r, we store a permutation π of $\{1,2,\ldots,d\}$ that keeps track of the original order in D of each leaf of PT_r. Namely, $\pi(i)$ is the index in D of the ith lexicographically smallest string in PT_r. Clearly, storing π requires $d\log d + O(d)$ bits. Figure 3 shows most of the data structures above built on the set of strings $D = \{\mathtt{abcc}, \mathtt{accb}, \mathtt{baca}, \mathtt{caac}, \mathtt{cbcc}\}$.

Candidate strings obtained by deleting a symbol. The identification of candidate strings for deletion is an easy task. Indeed, we observe that there are just p possible candidate strings obtainable from $P[1,p]$ by deleting one of its symbol. Thus, we simply consider any string $P[1,i]\cdot P[i+2,p]$ as a candidate string. However, any of these strings is reported only after having checked that it actually belongs to D. As said above, for the moment we assume that this non-trivial task can be done in $O(1)$ (amortized) time per string.

Candidate strings obtained by inserting or substituting a symbol. Identifying candidate strings for insertion or substitution of a symbol is an easy task whenever the alphabet has constant size. In this case there are, indeed, $O(\sigma\cdot p) = O(p)$ candidate strings obtained by inserting or substituting any possible symbol of Σ in any position of P. This implies that the data structures above suffice for point 1 in Theorem 3[4]. Identifying insertions and substitutions with a larger alphabet is a much harder task, which requires an additional data structure. Our additional data structure follows the idea presented in [4] which allows us to reduce the number of candidate strings from $O(\sigma\cdot p)$ to $O(p+occ)$. However, our solution is forced to use more sophisticated arguments in order to achieve a space bounded in terms of the k-th order entropy. Given the set of strings D and the two patricia tries PT and PT_r, our first step consists in

[4] Recall that we are still assuming that we can check in $O(1)$ whether a candidate string belongs to D.

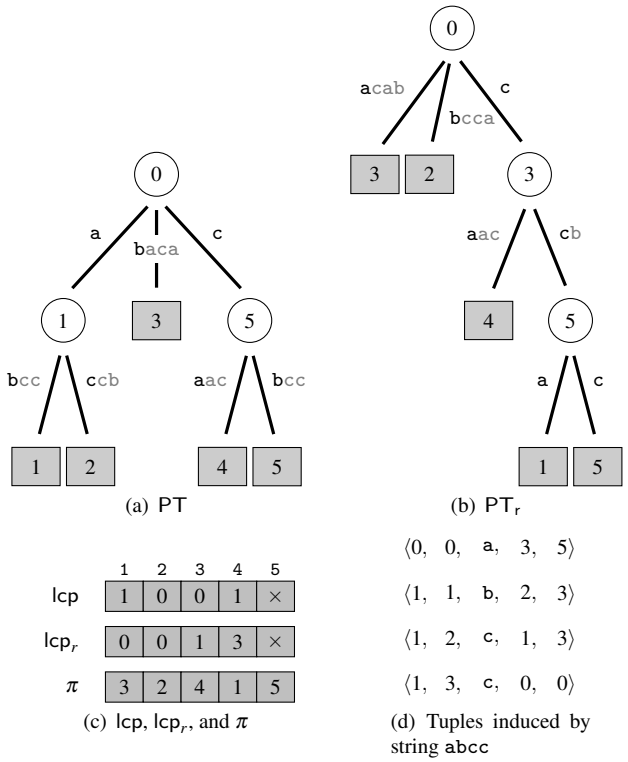

Fig. 3 The picture shows a running example for the set of strings $D = \{\texttt{abcc}, \texttt{accb}, \texttt{baca}, \texttt{caac}, \texttt{cbcc}\}$. Figures (a) and (b) show the patricia tries of, respectively, the strings in D and the strings in D written in reversed order. For each internal node we report the time of its visit in a preorder visit of the tree while for each leaf we report the identifier of the corresponding string in D. We report complete edges labels, even if a patricia trie stores only the first symbol of each label. Figure (c) reports lcp, lcp$_r$, and π. Figure (d) shows the four tuples induced by the string abcc.

building a set T of tuples. For each string S in D of length s, we consider each of its factorizations of the form $S = S[1,i] \cdot c \cdot S[i+2,s]$. For each of them, we add to T the tuple $\langle \mathsf{np}, i, c = S[i+1], s-(i+1), \mathsf{ns} \rangle$ where np (resp. ns) is the index of the node in PT (resp. PT$_\mathsf{r}$) whose locus has the longest common prefix with $S[1,i]$ (resp. $S[i+2,s]$ reversed). Observe that the cardinality of T is at most n, since we add at most s tuples for a string S of length s. Figure 3(d) shows the four tuples induced by string $S = \texttt{abcc}$. For example, the second tuple is equal to $\langle 1, 1, \texttt{b}, 2, 3 \rangle$ since the locus of node 1 in PT has the longest common prefix with $S[1,1] = \texttt{a}$, the locus of node 3 in PT$_\mathsf{r}$ has the longest common prefix with $S[3,4] = \texttt{cc}$ reversed, and $S[2] = \texttt{b}$.

The set T contains enough information to allow the identification of all the candidate strings. In the following we consider only insertions since substitutions are solved similarly. For insertions we consider all the factorizations of P having the form $P = P[1,i] \cdot P[i+1,p]$. For each of them, we identify the (highest) nodes np$_i$ and ns$_{i+1}$ in PT and PT$_\mathsf{r}$ that are prefixed respectively by $P[1,i]$ and $P[i+1,p]$ reversed. Clearly, identifying all these nodes for all the factorizations of P requires $O(p)$ time.

The key observation to identify candidate strings is the following: If there exists a tuple $\langle \mathsf{np}_i, i, c, p-i, \mathsf{ns}_{i+1} \rangle$ in T, then the string $S = P[1,i] \cdot c \cdot P[i+1,p]$ belongs to D and, obviously, has distance one from P.[5] As an example, consider the pattern $P = \mathtt{acc}$. The node $\mathsf{np}_1 = 1$ has the longest common prefix with $P[1,1] = \mathtt{a}$ in PT and the node $\mathsf{ns}_2 = 3$ has the longest common prefix with $P[2,3] = \mathtt{cc}$ reversed in PT_r. Since the triple $\langle 1,1,\mathtt{b},2,3 \rangle$ belongs to T, the string $P[1,1] \cdot \mathtt{b} \cdot P[2,3] = \mathtt{abcc}$ has distance one from P and belongs to D.

Our data structure is built on top of T and allows us to easily identify the required tuples (and without requiring to store T explicitly). We notice that there may exist several tuples of the form $\langle \mathsf{np}, i, \star, \mathsf{ns}, i' \rangle$. These groups of tuples share the same four components np, i, ns and i', and differ just for the symbol c. In order to distinguish them, we arbitrarily rank tuples in the same group and we assign to each of them its position in the ranking. We build a data structure that, given the indexes np and ns of two nodes, two lengths i and i' and rank r, returns the symbol c of the rth tuple of the form $\langle \mathsf{np}, i, \star, \mathsf{ns}, i' \rangle$ in T. The data structure is allowed to return an arbitrary symbol whenever such a tuple does not exist. The use of such a data structure to solve our problem is simple. For each factorization $P[1,i] \cdot P[i+1,p]$ of P, we query the data structure above several times by passing the parameters np_i, i, $p-i-1$, ns_{i+1} and r. The value of r is initially set to 0 and increased by 1 for each of the subsequent queries. After every query, we check if the string $S = P[1,i] \cdot c \cdot P[i+1,p]$ belongs to D, where c is the symbol returned by the data structure. We pass to the next factorization as soon as we discover that either the string S does not belong to D or symbol c has been already seen for the same factorization. Both these conditions provide the evidence that no tuple $\langle \mathsf{np}_i, i, \star, p-i-1, \mathsf{ns}_{i+1} \rangle$ with rank r or larger can belong to T. It is easy to see that the overall number of queries is $O(p+occ)$.

We are now ready to present a data structure to index T as described above that requires $O(1)$ time per query and uses entropy bounded space. The first possible compressed solution consists in defining a function F which is then represented by using the solution of Theorem 1. For any tuple $\langle \mathsf{np}, i, c, \mathsf{ns}, i' \rangle$ having rank r in T, we set $F(\mathsf{np}, i, \mathsf{ns}, i', r)$ equal to c. Queries above are solved by appropriately evaluating function F. According to Theorem 1, each query is solved in constant time. As far as space occupancy is concerned, we observe that F is defined for at most n values and that any symbol of any string in D is assigned at most once. Thus, by combining these considerations with Theorem 1, it follows that the representation of F requires at most $nH_0 + O(\frac{n(H_0+\log\log n)\log\log n}{\log n})$ bits. A boost of this space complexity to nH_k is obtained by defining several functions F, one for each possible context of length k. Here $k = o(\log_\sigma n)$ is an arbitrary but fixed parameter. The function F_{cntxt} is defined only for tuples $\langle \mathsf{np}, i, c, \mathsf{ns}, i' \rangle$ where the symbol c is preceded by the context cntxt in the string that induced the tuple. By summing up the cost of storing the representations of these functions, we have that the space occupancy is bounded by $nH_k + O(\frac{n(H_k+\log\log n)\log\log n}{\log n})$ bits for the fixed $k = o(\log_\sigma n)$. Notice that splitting F in several functions is not an issue for our aim. In the algorithm above, indeed, we can always query the correct function since we are aware of the correct context.

[5] Observe that similar considerations hold also for substitutions with the difference that we skip the ith symbol in factorizations of the form $P = P[1,i-1] \cdot P[i] \cdot P[i+1,p]$.

Checking candidate strings. We are left to explain how to establish, in constant time, whether a candidate string belongs to D. Observe that any candidate string has the form $S = P[1,i] \cdot P[i+2,p]$ in case of deletion, $S = P[1,i] \cdot c \cdot P[i+1,p]$ in case of insertion, or $S = P[1,i] \cdot c \cdot P[i+2,p]$ in case of substitution, for some symbol c and index i. One of the issues behind this task is the fact that S may not fit in a constant number of memory words and, thus, it cannot be managed directly in constant time. For this aim Karp-Rabin function kr is used to create small sketches of the strings in D that fit in $O(1)$ memory words and that uniquely identify each string. Observe that the signatures assigned by function kr are values smaller than M and, thus, each of them fits in $O(1)$ words of memory.

Once we have these perfect signatures, we use a minimal perfect hash function to connect each signature to the corresponding string in D. Let D_{kr} be the set of signatures assigned by kr to strings in D (i.e., $D_{\mathsf{kr}} = \{\mathsf{kr}(S) \mid S \in D\}$). We construct a minimal perfect hash function mph that maps signatures in D_{kr} to the first d integers. Lemma 3 guarantees $O(1)$ evaluation time by requiring $O(d)$ bits of space. As satellite data, the entry for the string S stores in $\log d + O(1)$ bits the index of the leaf in $\mathsf{PT_r}$ that corresponds to S reversed. Clearly, if S belongs to D, $\mathsf{mph}(\mathsf{kr}(S))$ gives us in constant time the index of S reversed in $\mathsf{PT_r}$ while $\pi(\mathsf{mph}(\mathsf{kr}(S)))$ reports the index of S in PT. It is worth noticing that the result of these operations are meaningless whenever S does not belong to D.

The checking of candidate strings requires a preprocessing phase shared among all the candidate strings. Firstly, we compute in $O(p)$ time the Karp-Rabin signatures of all prefixes and suffixes of P. In this way, the signature of any candidate string S can be computed in constant time by appropriately combining two of those signatures (see Lemma 2). Then, we identify a leaf pleaf in PT that shares the longest common prefix with P. Similarly, we identify a leaf sleaf in $\mathsf{PT_r}$ having the longest common prefix with P reversed. Given the properties of patricia tries and our succinct representation, identifying these two leaves costs $O(p)$ time.

The check for the single candidate string $S = P[1,i] \cdot c \cdot P[i+1,p]$ obtained by inserting symbol c in the $(i+1)$th position is done as follows[6]. We compute in constant time the values $k = \pi(\mathsf{mph}(\mathsf{kr}(S)))$ and $k' = \mathsf{mph}(\mathsf{kr}(S))$. Then, we have to check that the candidate string S is equal to the string S_k in D. Instead of comparing S and S_k symbol by symbol, we exploit the fact that S and S_k coincide if and only if the following three conditions are satisfied:

- $\mathsf{lcp}(k, \mathsf{pleaf})$ is at least i;
- $\mathsf{lcp}_r(k', \mathsf{sleaf})$ is at least $p - i$;
- $(i+1)$th symbol of S_k is equal to c.

Clearly, these three conditions are checkable in constant time. The $O(p)$ preprocessing time is amortized over the $O(p + occ)$ candidate strings.

Finding Top-k strings. As we mentioned in the introduction, our solution can be extended to support an additional operation which has interesting practical applications. Assume that each string S_i in D is assigned a score $c(S_i)$. For example, the score could

[6] Checks for other types of errors are done in a similar way.

establish the relative importance of any string with respect to the others. It is possible to extend our solution in order to support the extra operation $\mathsf{Top}(P[1,p],k)$ that reports the k highest scored strings in D having edit distance at most one with P. This operation is solved in $O(p+k\log k)$ time. We assume that values $c()$ are available for free. Notice that we can easily avoid this assumption by storing in $d\log d+O(d)$ bits the ranking of strings in D induced by $c()$.

We first present a simpler $O((p+k)\log k)$ time algorithm which is, then, modified in order to achieve the claimed time complexity. We said above that an arbitrary rank is assigned to tuples in T belonging to the same group (namely, tuples of the form $\langle \mathsf{np}, i, \star, \mathsf{ns}, i' \rangle$ that differ just for the symbol $\star$). Instead, this algorithm requires that the assigned ranks respect the order induced by $c()$. Namely, lower ranks are assigned to tuples corresponding to strings with higher values of $c()$. The searching algorithm is similar to the previous one. The main difference is in the order in which the factorizations of $P[1,p]$ are processed. The algorithm works in steps and keeps a heap. The role of the heap is to keep track of the top-k candidate strings seen so far. Each factorization is initially considered *active* and becomes *inactive* later in the execution. Once a factorization becomes inactive, it is no longer taken into consideration. Each factorization also has an associated score which is initially set to $+\infty$. At each step, we process the active factorization with the largest score. We query function F with the correct value of r for the current factorization. Let S be the candidate string identified by resorting to F. If S does not belong to D, the current factorization becomes inactive and we continue with the next factorization. Otherwise, we insert S into the heap with its score $c(S)$ and we decrease the score associated with the current factorization to $c(S)$. At each step we also check the number of strings in the heap. If there are $k+1$ strings in the heap, we remove the string with the lowest score and we declare the factorization that introduced that string inactive.

Notice that, apart from the first k steps, in each step a factorization becomes inactive. Since there are $O(p)$ factorizations, our algorithm performs at most $O(p+k)$ insertions into a heap containing at most k strings. Thus, the claimed time complexity easily follows. The improvement is obtained by observing that most of the time (i.e., $O(p\log k)$) is spent on inserting the first string of each factorization into the heap. This is no longer necessary if we use the following strategy. We first collect the first string of each factorization together with its score and we apply the classical linear time selection algorithm to identify the k-th smallest score. This step costs $O(p)$ time. We immediately declare the $p-k$ factorizations whose strings have a smaller score inactive. We insert the remaining k strings into the heap and we use the previous algorithm to complete the task. The latter step costs now $O(k\log k)$ time, since we have just k active factorizations.

5 A Bwt-based solution

The term $d\log d$ and the factor 2 multiplying the H_k term in the space bound of Theorem 3 may be too large for some applications.

In this section we provide a solution that is able to overcome this limitation at the cost of (slightly) increasing the query time.

Formally, we prove the following theorem.

Theorem 4 *Given a set $D = \{S_1, S_2, \ldots, S_d\}$ of d strings of total length n drawn from an alphabet $\Sigma = [\sigma]$, $\sigma \leq n$ and let w be the size of a memory word, there exists an index requiring $nH_k + \rho$ bits of space for any $k \leq \alpha \log_\sigma n$ and $0 < \alpha < 1$ that, given any pattern $P[1,p]$ reports all the occ strings in D having edit distance at most one with P in*

1. *$O(p \log_\sigma n \log\log n + occ)$ worst-case time for $\sigma = O(\mathtt{poly}(w))$ with redundancy $\rho = O(n \frac{\log\sigma}{\log\log n}) + o(n)$ bits;*
2. *$O(p \log\log_w \sigma + occ)$ worst-case time for larger σ with redundancy $\rho = o(n)(1 + H_k) + O(n \log(\log_\sigma n \log\log n))$ bits.*

This solution uses a completely different approach with respect to the previous one and solves the problem by building a collection of compressed permuterm indexes [22] on the dictionary D. More precisely, we divide the strings in D into subsets based on their lengths and we build a compressed permuterm index R_ℓ for each set D_ℓ, where D_ℓ denotes the subset of strings in D of length ℓ.[7] This solution introduces several possible trade-offs but, for simplicity, we report in Theorem 4 only the most interesting ones.

The compressed permuterm index [22] is a compressed index for dictionaries of strings based on the Burrows-Wheeler Transform (Bwt). Among other types of queries, it solves efficiently the PrefixSuffix(P,S) query which is useful for our problem. This query, given a prefix P and a suffix S, identifies all the strings in the dictionary having P as prefix and S as suffix. Below we resort to a slightly different variant of the compressed permuterm index. The main difference is the sorting strategy used to obtain the underlying Burrows-Wheeler Transform (Bwt) [13]. In [22] a text is obtained by concatenating the strings in the dictionary by using a special symbol # not in Σ as separator. Then, all the suffixes of this text are sorted lexicographically to obtain the rows of the Burrows-Wheeler matrix. In our variant we first append the symbol # to each string, then we construct the (conceptual) matrix M by lexicographically sorting all the cyclic rotations of all the strings in the set. Since every row is a rotation of a single string from the dictionary it is guaranteed that any row contains only symbols belonging to the same string. This fact turns out to be useful below when we will define parent and depth operations on a proper (conceptual) trie. This different construction of the Burrows-Wheeler transform was already implicitly used by Ferragina and Venturini [22] and simulated at query time by means of function jump2end (see [22] for more details). Figure 4 shows this variant of the Burrows-Wheeler Transform for the cyclic rotations of the strings in $D_4 = \{\mathtt{abcc}, \mathtt{accb}, \mathtt{baca}, \mathtt{caac}, \mathtt{cbcc}\}$. A query PrefixSuffix$(P,S)$ can be easily solved by searching the pattern $S\#P$ with the standard Backward_search [22]. The procedure returns the range of rows of M that are prefixed by $S\#P$ which are exactly all the strings in the dictionary that are both prefixed by P and suffixed by S.

Given a pattern $P[1,p]$, we solve our problem by querying only three compressed permuterm indexes: R_{p-1} for deletions, R_p for substitutions and R_{p+1} for insertions.

[7] We notice that the number of distinct lengths and, thus, compressed permuterm indexes is $O(\sqrt{n})$.

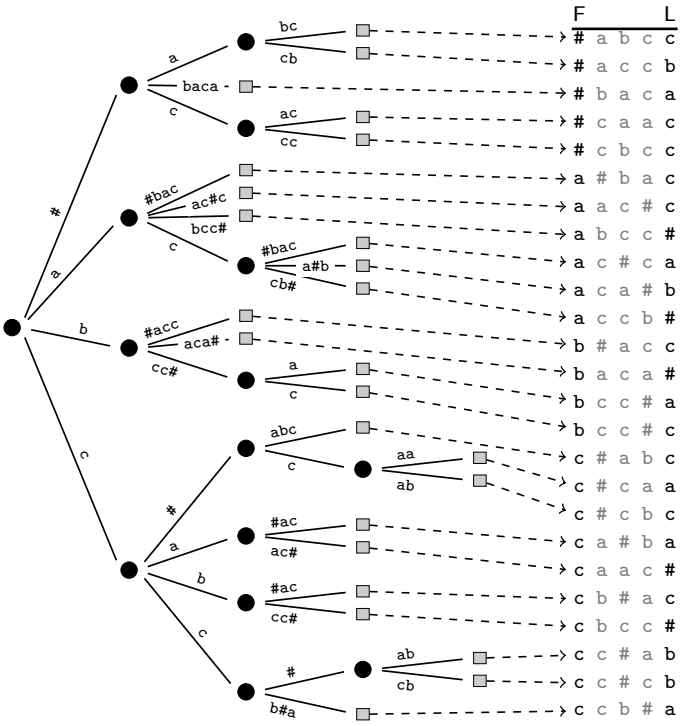

Fig. 4 The matrix M_4 for our variant of the Burrows-Wheeler Transform Bwt (right) which is obtained by sorting lexicographically all the cyclic rotations of strings in the set $D_4 = \{\texttt{abcc}, \texttt{accb}, \texttt{baca}, \texttt{caac}, \texttt{cbcc}\}$. The resulting Burrows-Wheeler Transform is the last column of this matrix (i.e., $\mathsf{Bwt}_4 = \mathsf{L}$). The Figure shows also the compact trie built on all these cyclic rotations. Dashed arrows show the existing relation between T_4 and M_4.

In the following we will only describe the solution for insertion since deletion and substitution are solved in a similar way. The basic idea behind our searching algorithm is the following. For each cyclic rotation $P_i = P[i,p]\#P[1,i-1]$ of $P[1,p]\#$, we use the compressed permuterm index R_{p+1} to identify the range of rows (say, $[l,r]$) of M_{p+1} that are prefixed by P_i, if any, where M_{p+1} is the matrix for the strings in D_{p+1}. We observe that having that range suffices for identifying the strings in D obtained by inserting a symbol at the ith position of P. These symbols are, indeed, the ones contained in $\mathsf{Bwt}_{p+1}[l,r]$, where Bwt_{p+1} is the Burrows-Wheeler Transform for the strings in D_{p+1}. However, we cannot compute all these ranges in a naïve way by searching each P_i separately using the backward search, since it would cost $\Omega(p^2)$ time. Thus, a faster solution has to efficiently move from rows prefixed by P_i to rows prefixed by P_{i-1}. This is achieved by augmenting the compressed permuterm index with a data structure that supports the two operations: parent and depth on a (conceptual) compact trie T_{p+1} which indexes all the cyclic rotations of strings in D_{p+1}. The trie for our set of five strings D_4 is shown in Figure 4. There exists a very strong relation between T_{p+1} and M_{p+1}: the locus of the ith leaf of T_{p+1} is equal to the ith row of M_{p+1}. Moreover, any internal node u of T_{p+1} is in correspondence with a

range of rows in M_{p+1} (namely, the rows corresponding to the leaves in the subtree rooted at u). These rows share a longest common prefix which is equal to the locus of u.

Let u be a node of the above trie corresponding a range of rows $[l,r]$ in M_{p+1}. The two operations are defined as follows:

1. $\mathsf{parent}(u)$ returns the range $[l',r']$ corresponding to the parent of the node u;
2. $\mathsf{depth}(u)$ returns the length of the locus of node u.

It is possible to support both these operations in $O(\log_\sigma n \log\log n)$ time by requiring $O(\hat{n}\frac{\log\sigma}{\log\log n})$ bits of additional space when $\sigma = O(\mathtt{poly}(w))$ [31], where $\hat{n}$ is the total size of the indexed dictionary.

Our solution works in two phases. In the first phase, it identifies the range of rows of Bwt_{p+1} sharing the longest common prefix with $P_0 = \#P[1,p]$. This is done by using the following strategy. We search P_0 backwards. At any step j, we keep the following invariant: $[l_j,r_j]$ is the range of all rows of Bwt_{p+1} prefixed by q_j, where q_j is the longest prefix of $P_0[p-j+2,p+1]$ that is a prefix of at least one row of Bwt_{p+1}. We also keep a counter ℓ that tells us the length of q_j. Notice that it may happen that a backward step from $[l_j,r_j]$ with the next symbol $P[p-j+1]$ returns an empty range. In this case, we repeatedly enlarge the range $[l_j,r_j]$ via parent operations until the subsequent backward step is successful. The value of ℓ is kept updated by increasing it by one after every successful backward step or by setting it equal to the value returned by depth after every call to parent. This approach has been already used to compute the so-called matching statistics [29].

Similarly, the second phase matches rotations of $P\#$ backwards. The main difference is given by the fact that the starting range and the value of ℓ are the ones computed at the end of the previous phase. At each step i, we identify the range of rows $[l_i,r_i]$ that share the maximal common prefix with $P[i,p]\#P[1,p]$. The correctness of each step follows from the following fact.

Fact 2 *The range of rows prefixed by $P_i = P[i,p]\#P[1,i-1]$ is non-empty if and only if the value of ℓ reaches $p+1$ or $p+2$.*

Proof In the former case, the obtained range $[l_i,r_i]$ is clearly the range of rows prefixed by $P_i = P[i,p]\#P[1,i-1]$. In the latter case[8], we need one additional step. In order to identify $P_i = P[i,p]\#P[1,i-1]$, we enlarge the range $[l_i,r_i]$ via parent operation, apply depth operation on it and check whether the returned value equals $p+1$. If that is the case, then the range of rows prefixed by $P_i = P[i,p]\#P[1,i-1]$ is the enlarged range. Otherwise, it is the original range. □

Even if this solution works for any alphabet size, we state its time and space complexities in point 1 of Theorem 4 for $\sigma = O(\mathtt{poly}(w))$ only. The overall time complexity of these two phases is $O(p\log_\sigma n\log\log n)$, since we have at most $2p$ calls to parent and depth which dominate the cost of the $O(p)$ calls to rank in the backward search. The overall space occupancy is $nH_k + o(n)$ bits for the compressed

[8] Notice that this case occurs only when $P_iP[i] \in D$. In order to properly deal with this case, the value of ℓ is not increased after a successful backward step if it already reached the maximal value $p+2$.

permuterm indexes and $O(n\frac{\log\sigma}{\log\log n})$ bits for the data structures to support parent and depth operations. This proves Point 1 of Theorem 4.

A better solution for larger alphabet sizes is reported in point 2 of Theorem 4. This solution is obtained by showing that, if we are allowed to use more space, a faster solution is possible. More precisely, we can improve the time of parent (for all cases) and depth (for the case of large depths) by augmenting every permuterm index R_ℓ with some auxiliary data structures:

1. The operation parent can be supported in constant time using $O(\hat{n})$ additional bits of space. This is feasible by using Sadakane's compressed suffix tree [32] on top the permuterm index R_ℓ.
2. The operation depth can be supported in constant time using $O(\hat{n}\log t)$ bits of space when the string depth is at least $\ell - t$, for some parameter t. For this aim, we resort to a table Δ_ℓ which stores $\log(t+1)$ bits per node. For any node u, we store the difference between the depth of the node u and p whenever the string depth of u is at least $\ell - t$. Otherwise, we store a special symbol indicating that the string depth of u is less than $\ell - t$.

Now that we have a constant time parent operation, the depth operation remains as the only bottleneck for achieving faster query time. Indeed, for larger alphabets, each depth operation requires $O(\log_\sigma n(\log\log n)^2)$ time and uses $O(\hat{n}\frac{\log\sigma}{\log\log n})$ bits of additional space [31]. To circumvent this, we introduce a lazy strategy that computes the correct value of ℓ only whenever its value may be $p+1$ or $p+2$ (i.e, we have a match), thus, avoiding most of the calls to depth operation. Assume that the compressed suffix tree supports the depth operation in time t. Instead of performing depth operation after each parent, the algorithm keeps track of the last node u obtained by parent operation with an associated variable d_u, which may be undefined. The value is always the depth of u whenever u's depth is at least $p+1-t$, but may be undefined otherwise. Every time we perform a parent operation, we use table Δ_ℓ to try to compute in constant time the value of d_u, but we set it to undefined whenever Δ_ℓ does not contain its depth (i.e., Δ_ℓ returns the special symbol). The algorithm also keeps track of the number t' of successive backward step after the last parent operation. This way, if d_u is defined, we can compute ℓ as $d_u + t'$, this is because every backward step after the last parent operation have increased it by one. Instead, if d_u is undefined, at least t backward steps are required for ℓ to be at least $p+1$. Thus, we compute the value d_u with a depth operation as soon as t' becomes equal to t. This way, we can check whether $\ell = d_u + t'$ is at least $p+1$.

It follows that a depth operation is computed only after exactly t successive backward steps from the last parent operation and that the result is kept for the subsequent successive backward steps. As a first consequence of this fact, the cost of depth operation can be amortized over the t successive backward steps. Another consequence is that the value d_u is always defined whenever $t' \geq t$. The correctness of the algorithm follows by observing that the range may correspond to a P_i only if either $t' \geq t$ or Δ_ℓ does not return the special symbol, and in both cases, d_u will be defined.

Point 2 of Theorem 4 is obtained by setting $t = O(\log_\sigma n(\log\log n)^2)$. Indeed, the backward search becomes the dominant time cost, namely, $O(p\log\log_w \sigma)$ time according to Theorem 2. To the space occupancy of Theorem 2 we have to add $O(n)$ bits

for the Sadakane's compressed suffix trees and $O(n\log t) = O(n\log(\log_\sigma n \log\log n))$ bits to store the tables Δ_ℓ.

6 Randomized solutions

In this section we provide two randomized solutions. The first one is a Monte Carlo solution which may report $O(\varepsilon)$ false positives in expectation (i.e., spurious non existing occurrences). Formally, we prove the following theorem.

Theorem 5 *Given a set $D = \{S_1, S_2, \ldots, S_d\}$ of d strings of total length n drawn from an alphabet $\Sigma = [\sigma]$, $\sigma \leq n$, there exists an index that, given any pattern $P[1,p]$, reports all the occ strings in D having edit distance at most one with P in $O(p + occ)$ randomized time. A query may report $O(\varepsilon)$ false positives in expectation (i.e., spurious non existing occurrences), for any parameter ε with $0 < \varepsilon < 1$. The index has size $nH_k + O(\frac{n(\log\log n + \log\sigma)\log\log n}{\log n}) + O(d\log\frac{1}{\varepsilon})$ bits, for any fixed $k = o(\log_\sigma n)$.*

The second solution is a Las Vegas solution which guarantees the space and time complexities reported in the following theorem.

Theorem 6 *Given a set $D = \{S_1, S_2, \ldots, S_d\}$ of d strings of total length n drawn from an alphabet $\Sigma = [\sigma]$, $\sigma \leq n$ and let w be the size of a memory word, there exists an index of size $nH_k + \rho + O(d\log\frac{1}{\varepsilon})$ bits, for any $k \leq \alpha\log_\sigma n$ with $0 < \alpha < 1$ and any parameter ε with $0 < \varepsilon < 1$, such that, given any pattern $P[1,p]$ reports all the occ strings in D having edit distance at most one with P in*

1. *$O(p + occ)$ time with probability ε and redundancy $\rho = O(n)$ bits, for $\sigma = O(\texttt{poly}(w))$;*
2. *$O(p\log\log_w\sigma + occ)$ time with probability ε and redundancy $\rho = O(n) + o(n)(1 + H_k)$ bits, for larger σ and $\log\sigma = O(\log n/\log\log n)$.*

Notice that these solutions could provide strong probabilistic guarantees by setting $\varepsilon = \frac{1}{n^c}$ for some constant c at the expense of using $O(d\log n)$ more bits of space. In this case the first solution only returns $O(\frac{1}{n^c})$ spurious occurrences in expectation, while the second one guarantees that the query time holds with high probability.

Monte Carlo solution. In this paragraph we show how to derive a Monte Carlo solution from the result of Theorem 3 in Section 4. The possibility of returning spurious answers combined with the use of approximate membership data structures suffices for reducing the dominant term in the space of this solution from $2nH_k$ to nH_k without increasing the query time. Fix the parameter ε and build an approximate membership AM_ℓ for each subset of strings D_ℓ of length ℓ by fixing the false positive probability to $\frac{\varepsilon}{\ell}$. According to Lemma 4 the space of AM_ℓ is $O(\log\ell + \log\frac{1}{\varepsilon})$ bits for each string in D_ℓ. Overall the space used by all the approximate membership data structures is $O(d(\log\frac{n}{d} + \log\frac{1}{\varepsilon}))$ bits, which is $O(\frac{n\log\sigma\log\log n}{\log n} + d\log\frac{1}{\varepsilon})$ bits according to Fact 1. The presence of these data structures and the relaxed goal allow us to remove the compressed scheme of Lemma 1 and the permutation π

in the solution of Section 4. This reduces the space occupancy in Theorem 3 to $nH_k + O(\frac{n(\log\log n+\log\sigma)\log\log n}{\log n}) + O(d\log\frac{1}{\varepsilon})$ bits.

Because of the suppression of the compressed string representations, the correctness of the steps performed during the percolation of the patricia tries can no longer be established by comparing the searched string and labels on the traversed edges. However, the percolation never introduces any false negative. Indeed, if $\mathsf{lcp}(|P|,\mathsf{pleaf})$ is at least i, then np_i is correctly computed. Similarly, if $\mathsf{lcp}_r(|P|,\mathsf{sleaf})$ is at least $p-i$, then ns_{i+1} is also correctly computed. This is because we can always follow the correct edge thanks to the monotone minimal perfect hash function stored at each node. Notice that a tuple $\langle \mathsf{np}_i, i, c, p-i, \mathsf{ns}_{i+1}\rangle$ can exist in T only if $\mathsf{lcp}(|P|,\mathsf{pleaf}) \geq i$ and $\mathsf{lcp}_r(|P|,\mathsf{sleaf}) \geq p-i$. Thus, we conclude that the traversal of the trie introduces no false negatives.

Also although it is no longer possible to establish whether a candidate string belongs to D, we can use the approximate membership data structure to ensure that a non-existing string is reported with probability at most ε. The check for a single candidate string $S = P[1,i]\cdot c\cdot P[i+1,p]$ obtained by inserting symbol c in the $(i+1)$th position is done simply by querying AM_{p+1}. Notice that a single non-existing candidate is reported only with probability $\frac{\varepsilon}{\ell}$. Moreover, a second candidate $S = P[1,i]\cdot c'\cdot P[i+1,p]$ with $c' \neq c$ is checked only if the first candidate was reported as existing [9]. Thus, on expectation the number of false positive candidates obtained by inserting a symbol in the $(i+1)$th position is bounded by $O(\frac{\varepsilon}{\ell})$. This gives $O(\varepsilon)$ false positive candidates when summing up over all positions of insertion. Similar considerations can be used to deal with substitutions and deletions.

Las Vegas solution. We now present the Las Vegas solution of Theorem 6 which is obtained by introducing randomization in the solution of Section 5. The goal here is to remove the use of depth operation whose time complexity was the dominant cost in Theorem 4. Similarly to the previous randomized solution, we build an approximate membership data structure AM_ℓ for each set D_ℓ. As in Section 5 we use Sadakane's compressed suffix tree [32] to support parent operation by using a constant number of bits per symbol. Given a pattern $P[1,p]$, we show how to find insertions using the compressed permuterm index R_{p+1} in conjunction with the approximate membership data structure AM_{p+1}, the parent operation, and the decompression of strings with LF steps on Bwt_{p+1}. As for the other solutions, substitutions and deletions are solved with a similar approach. Recall from Section 5 that our goal is to identify the range of rows (say, $[l,r]$) of M_{p+1} that are prefixed by P_i, for each cyclic rotation $P_i = P[i,p]\#P[1,i-1]$ of $P[1,p]\#$. Indeed, the symbols to be inserted at position i of P are the ones contained in $\mathsf{Bwt}_{p+1}[l,r]$. As in Section 5 the solution identifies, for each i, the range of rows sharing the maximal common prefix with $P[i,p]\#P[1,p]$ by combining the use of Backward_search and parent operation. The solution in Section 5 establishes that one of these ranges contains answers for insertion by checking that

[9] If we have false positives then the same character may be checked and thus potentially reported twice. To avoid this case, we can use a dynamic hash table at query time which stores all the characters reported so-far. Whenever we find that a character has been already reported, then the query stops and does not report more characters, since a correct query answer can not return the same character twice at the same position.

depth operation returns a value at least $p+1$ (i.e., the length of the maximal common prefix above is at least $p+1$). We show here how to replace the use of depth operation with LF steps. Indeed, we use the function *LF* to extract symbols backwards from one position in each of these ranges to check whether the depth is at least $p+1$. Notice that doing this in a naïve way would require to extract $\Omega(p^2)$ symbols. In the following we show an approach that extracts only $O(p)$ symbols from ranges having depth at least $p+1$. Proper queries to AM_{p+1} are used to filter (most of the) ranges with a shorter depth.

More in detail, let $[l_i, r_i]$ denote the range sharing the maximal common prefix with $P[i,p]\#P[1,p]$ computed with the only use of Backward_search and parent operation. For each i, we query AM_{p+1} for the candidate string $P[1,i-1]\cdot c\cdot P[i,p]$ obtained by inserting the symbol c in ith position of P, where c is any symbol in $\mathsf{Bwt}_{p+1}[l_i,r_i]$. This is the preliminary filter that removes a range with depth smaller than $p+1$ with probability at least $1-\varepsilon/p$. Let $I=\{i_1,i_2,\ldots,i_t\}$, with $i_1<i_2,\ldots<i_t$, be set of indices for which AM_{p+1} returns true. A naïve approach would work as follows. For each index i in I, it starts to extract $p+2$ symbols with $p+2$ LF steps starting from any row in $[l_i,r_i]$ and computes the maximal common prefix between the extracted string and $P[i,p]\#P[1,p]$. By definition, the depth of the range coincides with the length of this common prefix. Unfortunately, this approach requires $O(p)$ LF steps for each of the $O(p)$ indices in I. In our solution we use the approach above starting from the smallest index i_1 to check that the range of rows $\mathsf{M}[l_{i_1},r_{i_1}]$ is prefixed by $P_{i_1}=P[i_1,p]\#P[1,i_1-1]$. If this check succeeds, then we know that for any other index $j>i_1$, the range of rows $\mathsf{M}[l_j,r_j]$ and $P[j,p]\#P[1,p]$ share a prefix of length at least $p+1-(j-i_1)$. Indeed, if we rotate the range of rows $\mathsf{M}[l_{i_1},r_{i_1}]$ by $j-i_1$ positions to the left, we get a range prefixed by $P[j,p]\#P[1,i_1-1]$ that shares a prefix with $P[j,p]\#P[1,p]$ of length at least $p+1-(j-i_1)$. Thus, we can check whether $\mathsf{M}[l_{i_2},r_{i_2}]$ is prefixed by $P_{i_2}=P[i_2,p]\#P[1,i_2-1]$ by performing only i_2-i_1+1 LF steps to check whether any row in $\mathsf{M}[l_{i_2},r_{i_2}]$ is preceded by the string $P[i_1,i_2-1]c$ where c is some symbol in $\mathsf{Bwt}_{p+1}[l_{i_2},r_{i_2}]$ (i.e., the row ends by $P[i_1,i_2-1]c$). If the check for i_1 fails (i.e., the range of rows $\mathsf{M}[l_{i_1},r_{i_1}]$ is not prefixed by $P_{i_1}=P[i_1,p]\#P[1,i_1-1]$), then the check for i_2 would require $p+2$ LF steps. This procedure is repeated for the subsequent indices in I.

To compute the number of LF steps required by this approach, it is convenient to split the set I in the two subsets I_a and I_b that contain indices that pass and fail the above check, respectively. Assume $I_a=\{i'_1,i'_2,\ldots,i'_{t'}\}$, with $i'_1<i'_2<\ldots<i'_{t'}$. The check for i'_1 asks to perform $p+2$ LF steps. The check for any other $i'_j>i'_1$ in I_a is done by making $i'_j-i'_{j-1}+1$ LF steps. Summing up over all the elements in I_a, this gives $O(p)$ LF steps, which, by using the data structure in Lemma 5, require $O(p)$ or $O(p\log\log_w\sigma)$ time depending on the alphabet size. Indices in I_b, instead, require $O(p)$ LF steps each. However, since the AM_{p+1} has a false positive probability $\frac{\varepsilon}{p+2}$, I_b contains on expectation $O(\varepsilon)$ indices, incurring $O(\varepsilon\cdot p)$ additional *LF* steps.

We have to address a small technical detail to conclude the proof of Theorem 5. Once we have determined the indices i such that the rows in $\mathsf{M}[l_i,r_i]$ are prefixed by $P_i=P[i,p]\#P[1,i-1]$, we need one more step. If $l_i<r_i$, we report all the symbols in $\mathsf{Bwt}_{p+1}[l_i,r_i]$ because we are sure that $S=P[1,i-1]\cdot c\cdot P[i,p]$ is a correct answer. On the other hand, if $r_i=l_i$, it may happen that the row $\mathsf{M}[l_i]$ equals $P[i,p]\#P[1,i-1]P[i]$.

In this case, there may exist a larger range of rows prefixed by $P[i,p]\#P[1,i-1]$ that lead to correct answers. This larger range is identified by taking the parent for the interval $[l_i,r_i]$ and by checking that the resulting new interval corresponds to a node of depth exactly $p+1$. If it is the case, we replace $[l_i,r_i]$ with the larger interval. Instead of computing the depth for that node, we associate one additional bit to mark every internal node of the trie that has depth exactly $p+1$. This solves the issue by adding $O(n)$ additional bits.

7 Conclusion

In this paper we described two different compressed solutions for searching with edit distance one in a dictionary of strings. The first solution requires $2H_k(S) + n \cdot o(\log\sigma) + 2d\log d$ bits of space for any fixed $k = o(\log_\sigma n)$. The query time is (almost optimal) $O(|P| + occ)$ time where occ is the number of strings in the dictionary having edit distance at most one with the query pattern P. The second solution further improves this space complexity but the time complexity grows to $O(|P|\log_\sigma n \log\log n + occ)$ or $O(|P|\log\log_w \sigma + occ)$ depending on the amount of redundancy, where w is the size of a memory word. Interestingly enough, the two solutions solve the problem at hand with two different approaches: the former is based on (perfect) hashing while the latter is based on the compressed permuterm index. Finally, we have shown how to introduce randomization in these solutions to derive Monte Carlo and Las Vegas solutions in order to either reduce the space occupancy or improve the query time of the deterministic solutions.

An interesting open problem asks for a deterministic approach that obtains simultaneously the time complexity of our first deterministic solution and the space complexity of the second one. Furthermore, it is still open whether one can design a solution that solves the problem in $O(|P| \cdot \log\sigma/w + occ)$ time. At the moment, there does not exist any solution achieving such a time complexity, even a non compressed solution (using say $O(n\,\texttt{polylog}(n))$ space).

Finally, building efficient dictionaries for edit distance d larger than 1 is still an open problem. However, the approaches we used in our two solutions are not easily extendible to efficiently solve query for higher edit distance. Indeed, we could just solve a query in $O(\sigma^{d-1}|P|^d + occ)$ time for edit distance d by resorting to the standard dynamic programming approach.

References

1. Amihood Amir, Dmitry Keselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text indexing and dictionary matching with one error. *Journal of Algorithms*, 37(2):309–325, 2000.
2. Jérémy Barbay, Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7(4):52, 2011.

3. Jrmy Barbay, Francisco Claude, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.
4. Djamal Belazzougui. Faster and space-optimal edit distance "1" dictionary. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 154–167, 2009.
5. Djamal Belazzougui. Improved space-time tradeoffs for approximate full-text indexing with one edit error. *Algorithmica*, pages 1–27, 2011.
6. Djamal Belazzougui and Gonzalo Navarro. New lower and upper bounds for representing sequences. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA)*, pages 181–192, 2012.
7. Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):23, 2014.
8. Djamal Belazzougui and Rossano Venturini. Compressed string dictionary look-up with edit distance one. In *Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 280–292, 2012.
9. Djamal Belazzougui and Rossano Venturini. Compressed static functions with applications. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 229–240, 2013.
10. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.
11. Gerth Stølting Brodal and Leszek Gąsieniec. Approximate dictionary queries. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, pages 65–74. Springer Verlag, 1996.
12. Gerth Stølting Brodal and Venkatesh Srinivasan. Improved bounds for dictionary look-up with one error. *Information Processing Letters*, 75(1-2):57–59, 2000.
13. Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
14. Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 30–39, 2004.
15. Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 91–100, 2004.
16. Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proceeding of the 19th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 235–246, 1992.
17. Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21:246–260, 1974.
18. Robert Mario Fano. On the number of bits required to implement anassociative memory. *Memorandum 61, Computer Structures Group, Project MAC*, 1971.
19. Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.

20. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
21. Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. *Theorectical Computer Science*, 372(1):115–121, 2007.
22. Paolo Ferragina and Rossano Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, 7(1):10, 2010.
23. Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.
24. Torben Hagerup and Torsten Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 317–326, 2001.
25. Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
26. Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
27. J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
28. Gonzalo Navarro and Veli Mäkinen. Compressed full text indexes. *ACM Computing Surveys*, 39(1), 2007.
29. Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *SPIRE*, pages 347–358, 2010.
30. Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 823–829, 2005.
31. Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):53, 2011.
32. Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
33. Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, 1999.
34. Andrew Chi-Chih Yao and Frances F. Yao. Dictionary look-up with one error. *Journal of Algorithms*, 25(1):194–202, 1997.